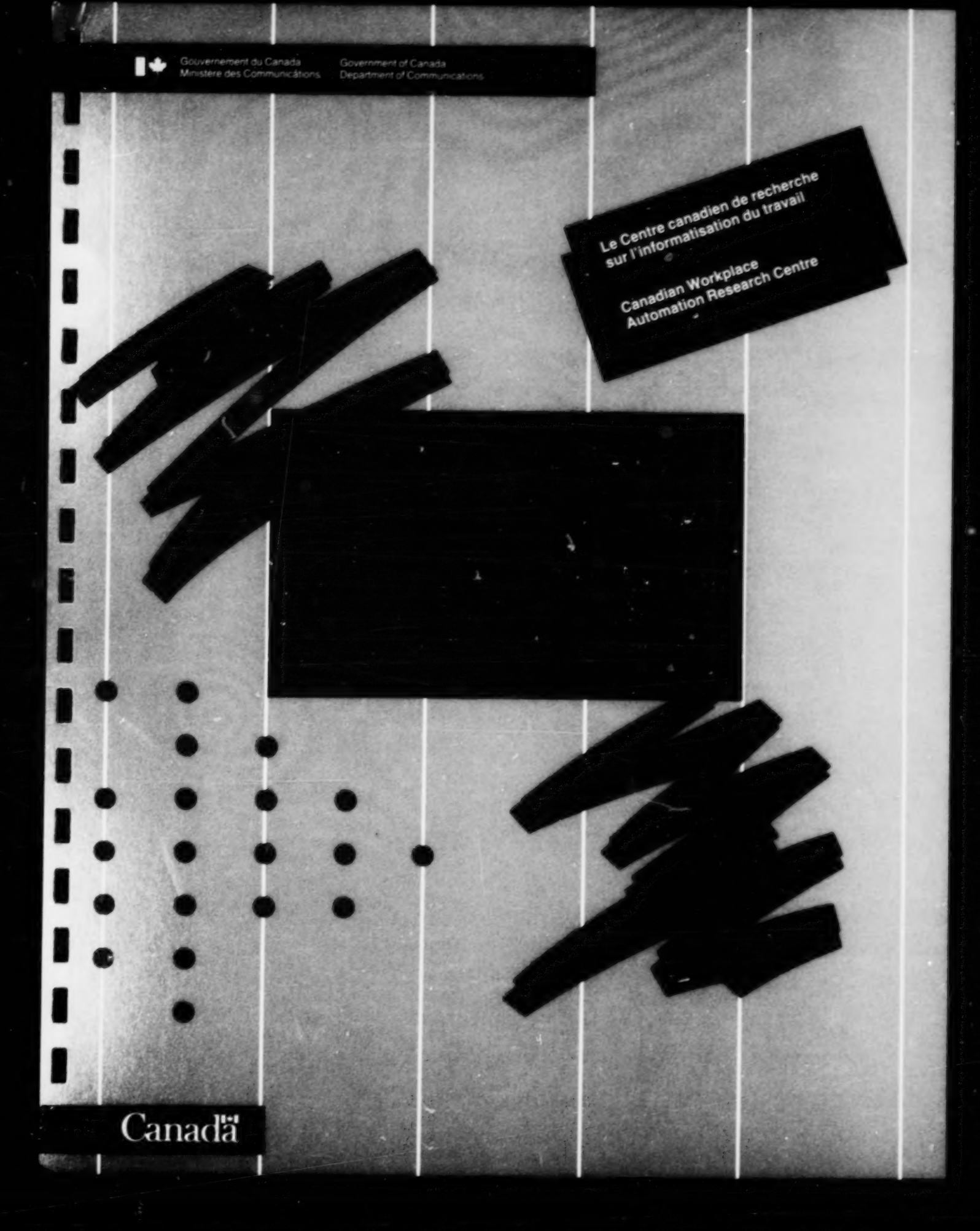

Gouvernement du Canada
Ministère des Communications
Government of Canada
Department of Communications
Le Centre canadien de recherche sur l'informatisation du travail
Canadian Workplace Automation Research Centre
Canada

RATP: UN NOUVEAU FORMALISME DE LA FAMILLE DES GRAMMAIRES D'UNIFICATION

Marc Dymetman

Centre Canadien de Recherches sur
l'Informatisation du Travail

Ministère des Communications Canada

Laval

Avril 1987

N° de cat.
ISBN

Les opinions exprimées dans ce rapport n'engagent que leur auteur.

* This report is available in English.

1 - INTRODUCTION

1.1 Présentation générale de RATP.

RATP est un formalisme d'écriture de Grammaires d'Unification intégrant de façon naturelle une composante qui lui donne la puissance d'un langage de programmation général. Cette composante "langage de programmation" a une structure analogue à un ensemble de clauses de Horn en Prolog pur, à cette différence près que ce sont des Dags (Directed Acyclic Graphs: Graphes Acycliques Orientés) au lieu de termes qui jouent le rôle de structures de données de base.

L'inclusion d'une telle composante ajoute à la modularité du formalisme grammatical, permettant la définition et la réutilisation faciles de groupes de traits grammaticaux. Elle permet aussi de clarifier les notions de conjonction, de disjonction, de négation des structures fonctionnelles, ainsi que d'introduire la possibilité d'évaluations retardées ("goal-freezing").

Un compilateur a été écrit, qui prend une grammaire RATP et produit du Code Prolog. Cette traduction est facilitée par un encodage simple des DAGS sous forme de termes Prolog et par l'applicabilité directe du mécanisme d'unification de Prolog au résultat de cet encodage.

Ce plongement de RATP dans Prolog présente l'avantage qu'on bénéficie ainsi d'un environnement de programmation puissant (dans ce cas MProlog) et que, compte tenu du parallélisme existant entre la structure d'un programme RATP et celle d'un programme Prolog, on peut faire communiquer les 2 langages d'une façon naturelle.

1.2 Grammaires d'Unification.

Ce que nous appelons ici du terme général de "Grammaires d'Unification" correspond en fait à un ensemble de formalismes grammaticaux, les plus connus étant : Lexical Functional Grammar <Bresnan, Kaplan 1983> Unification Grammar <Kay 1983>, PATR II <Shieber, 1984>.

Ces formalismes, bien que différents, partagent cependant certaines caractéristiques importantes :

- Les structures grammaticales sont vues comme des structures récursives associant des valeurs à des étiquettes, ces valeurs pouvant elles-même être des structures du même type.

- Ces structures, qui ont reçu diverses dénominations ("feature structures", "functional structures", "f-structures", etc) peuvent être vues, du point de vue formel, comme des cas particuliers du concept de "graphe étiqueté orienté sans cycles" ou DAG (pour "Directed Acyclic Graph").

- Dans ces structures, on est capable d'accéder à des valeurs des différents niveaux de "profondeur" à l'aide de "chemins d'accès" constitués de chaines d'étiquettes.

- Les sous-structures d'une structure peuvent être "partagées", c'est à dire qu'une même sous-structure est accessible par des chemins différents à partir d'une même structure au-dessus d'elle.

- Le mécanisme de base de calcul des structures grammaticales est un processus compositionnel, qui "prend" plusieurs structures plus élémentaires et les "assemble" en une structure plus complexe grâce à un mécanisme d'*unification*.

1.3 Parallélisme entre les grammaires d'unification et les grammaires logiques. DAGS et Termes.

Une autre approche influente actuellement dans le domaine des formalismes grammaticaux est connue sous le terme générique de "grammaires logiques".
Ce terme décrit l'ensemble des formalismes grammaticaux construits "autour du" langage PROLOG. Comme on le sait, ce langage a tout d'abord été conçu justement dans un but d'application à la description grammaticale. Il en est découlé des formalismes variés. Citons plus spécialement : les "grammaires de métamorphose" <Colmerauer 1978>, les "DCG" <Pereira, Warren, 1980>, les "grammaires d'extraposition" <Pereira 1981>, les "gapping grammars" <Dahl 1985>.

Les grammaires logiques ont beaucoup de points communs avec les grammaires d'unification. Dans les deux cas, le mécanisme d'unification de structures grammaticales joue un rôle fondamental. La différence entre les deux classes de formalismes tient essentiellement, en fait, au type de structure de données de base manipulée par ces systèmes: *DAGs* dans le cas des grammaires d'unification, *termes* dans le cas des grammaires logiques.

Alors que dans un DAG, les sous-structures sont accédées par la donnée d'une étiquette (ou d'une suite d'étiquettes), et peuvent être en nombre indéfini, dans un terme, les sous-structures se différencient par leur positions argumentaires, le nombre d'arguments d'un terme étant fixé une fois pour toutes. De plus la notion d'accès à une sous-structure "profonde" d'une structure est beaucoup moins mise en valeur dans le cas des termes que dans le cas des DAGS; la facilité à décrire de tels accès dans les grammaires d'unification conduit à un style de programmation différent qui semble présenter des avantages de clarté pour l'écriture de grammaires.

Partant de cette constatation que les deux classes de formalismes se distinguent essentiellement par leur structure de données de base, il est naturel de se demander :

1) Si les DAGs peuvent se coder comme termes et si les termes peuvent se coder comme DAGS.

2) Si le mécanisme d'unification de termes et le mécanisme d'unification de Dags peuvent se ramener à un même mécanisme fondamental.

A ces deux questions, la réponse est oui, comme la suite du rapport le montre plus en détail. Le fait que ces 2 propriétés soient vraies a alors la conséquence importante qu'il devient possible d'utiliser le mécanisme d'unification de Prolog pour implanter de façon efficace les grammaires d'unification, plus spécifiquement un nouveau formalisme de la classe des grammaires d'unification : le langage RATP.

1.4 RATP comme formalisme grammatical et comme langage de programmation.

RATP est un langage d'écriture de grammaires qui, au sein de la classe des "grammaires d'unification", se rapproche du formalisme PATR <Shieber 1984>. (Le nom du formalisme est un compromis entre cette réminiscence et un hommage à l'"Underground Parisien"), mais qui présente certaine innovations (prédicats généraux sur les structures).

Une grammaire RATP est constituée de 2 types de constructions textuelles:

- des règles de grammaire hors-contexte augmentées.
- des définitions de prédicats, sous la forme de clauses de Horn d'un type particulier

- Les règles de grammaire sont composées chacune :

1) d'un "squelette" hors contexte

2) d'un "corps" associé à ce squelette (!), comportant une suite d'équations et de prédicats qui doivent être vérifiés par les structures grammaticales associées aux nonterminaux et terminaux du squelette, ou par les sous-structures de ces structures. Ces sous-structures sont accédées par des chemins d'accès formés d'étiquettes ayant une signification grammaticale.

- Les définitions de prédicats sont données sous la forme de clauses de Horn, comme en PROLOG. La différence essentielle avec Prolog tient à ce que les objets auxquels les prédicats s'appliquent sont des "structures de traits" (des DAGs) alors qu'en Prolog ce sont des termes.

Le fait d'introduire des définitions de prédicats de ce type permet d'étendre considérablement la puissance du formalisme, car ces prédicats permettent de spécifier des classes de DAGs arbitrairement complexes, et d'utiliser ces classes pour résumer un ensemble de contraintes structurelles.

En fait, l'introduction de ces prédicats donne au formalisme la puissance d'un langage de programmation universel ressemblant à Prolog (voir l'exemple "multiplication" en section 2), ce qui est d'importance annexe ici, mais ce qui a la conséquence importante que les notions de conjonction, disjonction, négation s'éclairent d'un jour nouveau, et que certains concepts utiles de Prolog (comme le "freeze") peuvent être importés naturellement dans le formalisme.

Une autre conséquence de cette approche, peut être la plus importante, est que le formalisme met en plein lumière le fait que les "grammaires d'unification" et les "grammaires logiques" sont des variantes très proches d'un paradigme commun, un fait qui tend quelquefois à être obscurci par les différentes écoles de pensée.

2 - LE FORMALISME RATP.

Nous commençons par donner quelques exemples de structures produites par RATP en analyse[1], que nous commentons brièvement, puis nous décrivons la structure de la grammaire qui permet d'obtenir ces résultats.

2.1 Exemples de structures produites par RATP

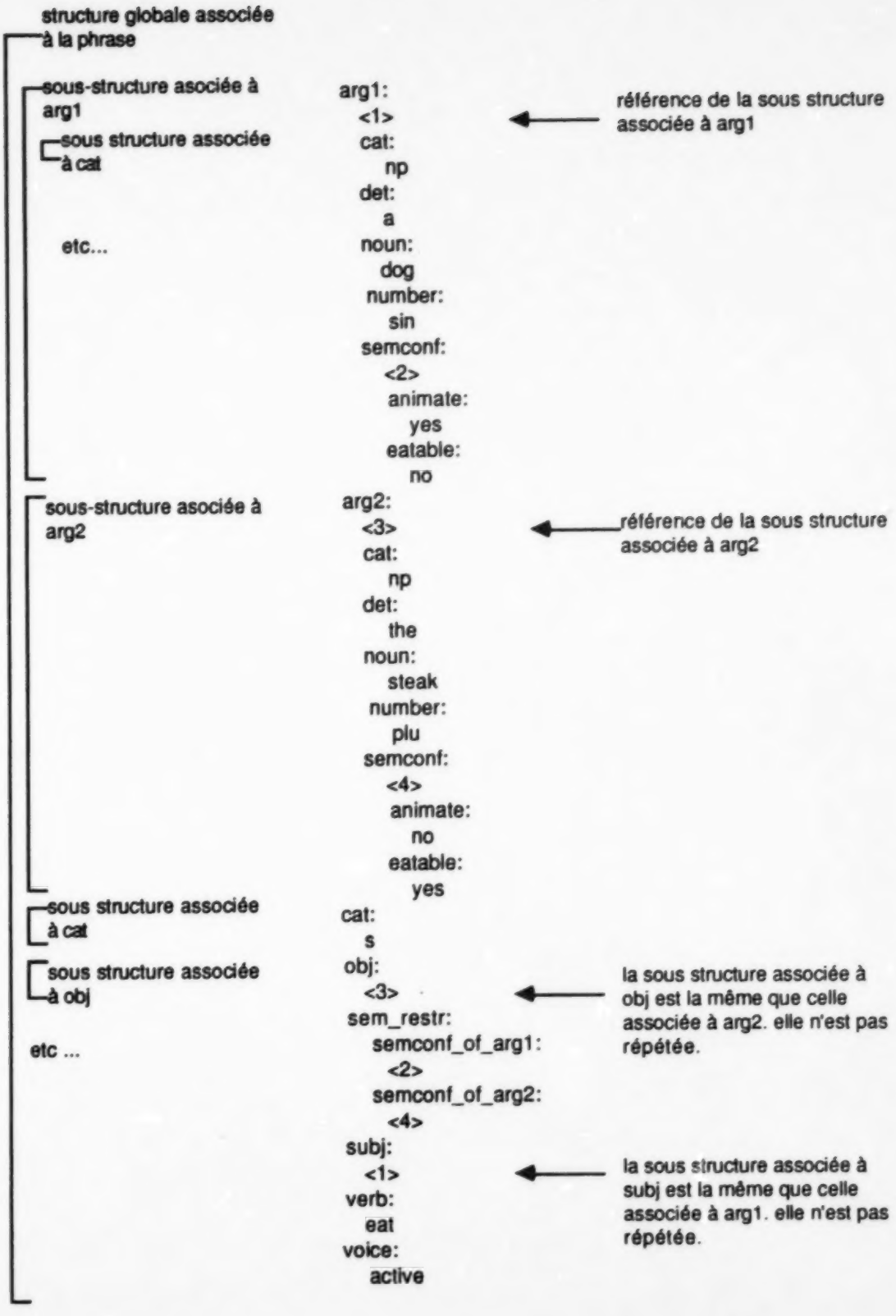
2.1.1. Exemples

Exemple 1 : Analyse de "a dog eats the steaks"

[a,dog,eats,the,steaks]

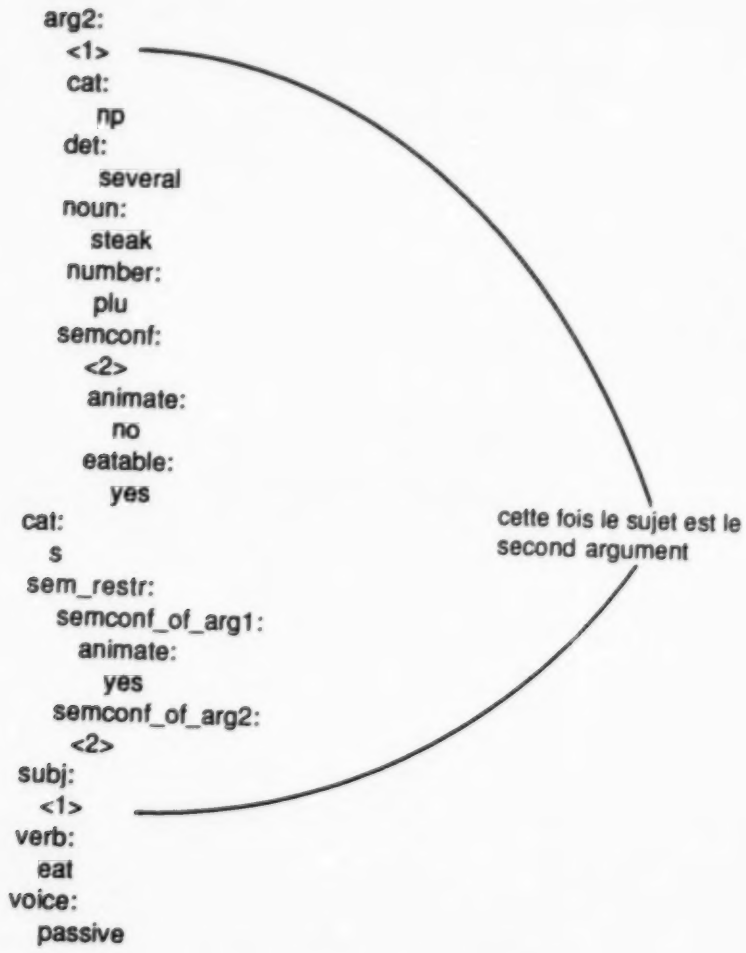
---

[1] Mentionnons que RATP peut s'utiliser en synthèse aussi bien qu'en analyse, une particularité que ce formalisme partage avec les DCG, par exemple. En RATP comme en DCG, la différence entre analyse et synthèse est simplement une différence dans la propagation des instanciations de variables.


structure globale associée à la phrase
sous-structure asociée à arg1
sous structure associée à cat
etc...
arg1:
<1>
cat:
np
det:
a
noun:
dog
number:
sin
semconf:
<2>
animate:
yes
eatable:
no
référence de la sous structure associée à arg1
sous-structure asociée à arg2
arg2:
<3>
cat:
np
det:
the
noun:
steak
number:
plu
semconf:
<4>
animate:
no
eatable:
yes
référence de la sous structure associée à arg2
sous structure associée à cat
cat:
s
sous structure associée à obj
obj:
<3>
la sous structure associée à obj est la même que celle associée à arg2. elle n'est pas répétée.
etc ...
sem_restr:
semconf_of_arg1:
<2>
semconf_of_arg2:
<4>
subj:
<1>
la sous structure associée à subj est la même que celle associée à arg1. elle n'est pas répétée.
verb:
eat
voice:
active

Exemple 2 : Analyse de "several steaks are eaten"

[several,steaks,are,eaten]



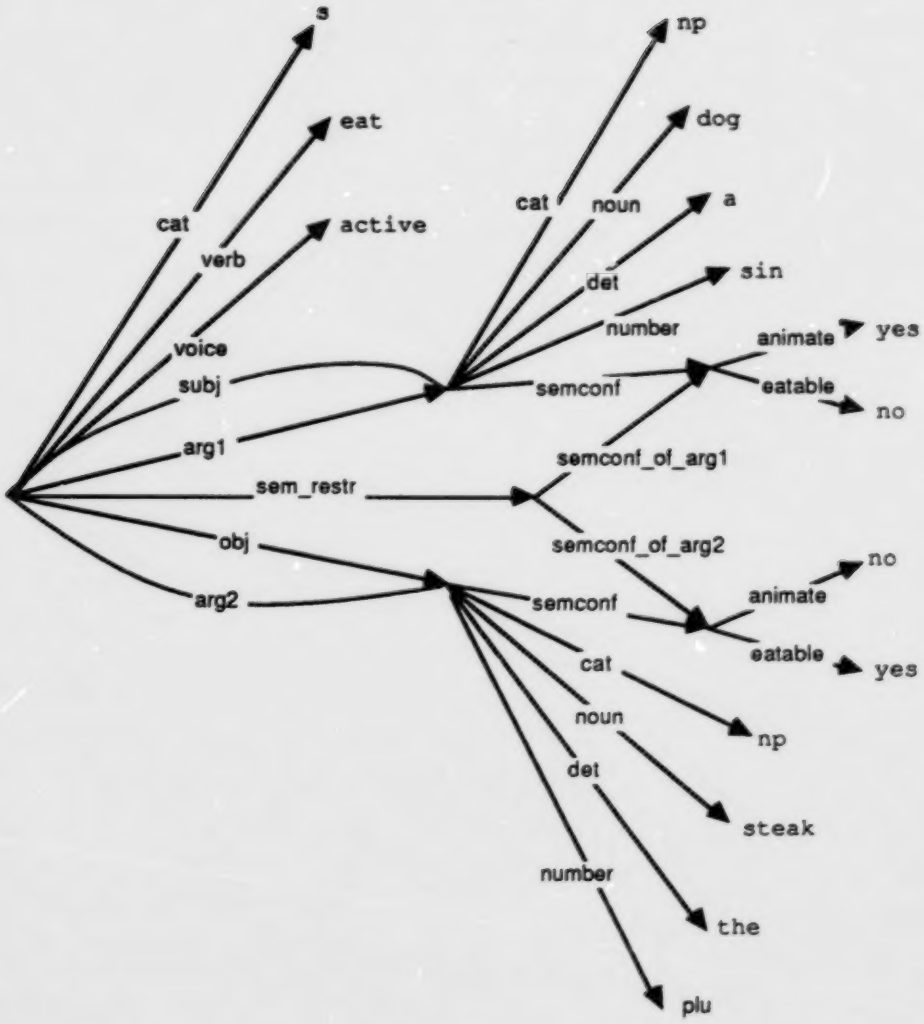
2.1.2 Explications des exemples

2.1.2.1 La notation

Considérons l'exemple 1. La structure donnée représente le résultat de l'analyse de la phrase : "a dog eats the steaks". Cette structure est un DAG[2] (Directed Acyclic Graph), qu'on pourrait représenter graphiquement de la façon suivante :

---

[2] pour être rigoureux, il faudrait distinguer entre *ground* Dags (Dags de base) et *nonground* (ou *variable*) Dags, mais c'est une distinction que nous ne ferons pas ici. La structure illustrée est en fait un *nonground* Dag, c'est à dire qu'elle représente une famile infinie de *ground* Dags.



Dag associé à "a dog eats the steaks"

Remarque préliminaire :

Dans la structure résultat de la page 7, l'ordre des étiquettes *arg1, arg2, cat, obj, subj, verb, voice* est l'ordre alphabétique. Cet ordre "vertical" entre les étiquettes n'a pas de signification particulière, et un ordre différent apparait dans la

représentation graphique de la page 9 (cat, verb...), pour des raisons de clarté graphique.

Le Dag représenté à la page 7 est la structure résultat de l'analyse de la phrase "a dog eat the steaks". Ce Dag a les propriétés suivantes :

- sa "cat" (catégorie) a pour valeur "s" ("s" est un Dag dégénéré cons titué simplement d'un symbole)

- son "verb" a pour valeur "eat" (idem)

- sa "voice" a pour valeur "active" (idem)

- son "subj" a pour valeur un Dag (non-dégénéré) dont la "cat" a pour
valeur "np", etc...

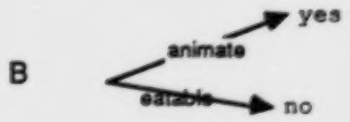
- son "arg1" a pour valeur le même Dag que son "subj". Dans la représentation graphique (p. 9), cela apparait par le fait que ces 2 Dags sont la même racine, alors que dans la représentation "textuelle" (p. 7), cela apparait par le fait que les étiquettes "arg1" et "subj" ont un indice de référence <1> identique associé. Par convention, on ne repète pas - au niveau du Dag imprimé - le contenu
de deux sous-Dags identiques, mais on utilise ces indices de référence pour alléger l'écriture[3], ceci seulement dans le cas de Dags non-dégénérés qui apparaissent de façon multiple.

La notion de chemin d' accès peut s'illustrer, sur cet exemple de la façon suivante : si on appelle A le Dag "global", alors le chemin

---

[3] Ces indices ont aussi une valeur "intensionnelle": deux objets coïndicés ont été unifiés à un moment donné et resteraient donc unifiés en cas d'utilisation ultérieure des structures. Ce fait n'apparaîtrait pas si on se contentait d'*imprimer* le contenu des 2 structures.

A.arg1.semconf (où "arg1" et "semconf" sont des étiquettes et A la constante qui représente le Dag global) est un "chemin d'accès" au Dag B, où B est la constante qui représente le Dag :



Le même objet B est aussi accédé par d'autres chemins, à savoir par les chemins A.subj.semconf A.sem_restr.semconf_of_arg1 (partage de sous-structures).

2.1.2.2 Remarques linguistiques.

La grammaire qui produit ces structures (grammaire présentée en section 2.2), bien que très rudimentaire, engendre des structures qui peuvent être lues à plusieurs niveaux, en particulier aux niveaux "syntaxique" et "logique". Au niveau syntaxique, la structure de l'exemple 1 ("a dog eat the steaks") fait apparaître un "verb", un "subj" et un "obj", alors qu'au niveau logique, elle fait apparaître un "verb", un "arg1" et un "arg2".

On remarque que, dans cet exemple, les sous-structures "accédées" par l'étiquette "subj" et par l'étiquette "arg1" sont identiques, ainsi que les sous-structures "accédées" par les étiquettes "obj" et "arg2", ce qui correspond au cas d'un verbe transitif à la voix active : le sujet syntaxique et le 1er argument logique se confondent, l'objet syntaxique et le 2ème argument logique se confondent[4].

---

[4] On remarquera l'analogie avec les structures "dépendancielles" du GETA, d'une part, et avec les structures fonctionnelles de Lexical Functional Grammar, d'autre part. Ici, comme dans ces deux cas, les structures associées aux arguments sont de même nature que les structures associées aux fonctions syntaxiques. En fait, ces structures

Far contre dans l'exemple 2 - "several steaks are eaten" - au niveau logique, le seul argument présent est l'arg2, et il est réalisé, au niveau syntaxique, dans la fonction sujet "subj". Cela correspond à l'un de cas de figure pour la voix passive, l'autre cas de figure correspondant à la présence d'un arg1 exprimé au niveau syntaxique par la fonction "by_phrase".

2.2 La Grammaire RATP qui produit ces exemples.

2.2.1 La Grammaire.

Voici, à titre d'illustration, la grammaire RATP (très élmentaire) qui permet l'analyse des exemples de la section 2.1.

A. Règles Hors-Contexte Augmentées.

```
S -> NP VP,
S.cat=s,
NP.cat=np,
VP.cat=vp,                                                    (1)
number_agreement(NP,VP),
S.subj=NP,
verbal_head_features_sharing(S,VP),
voice_treatment(S);

VP -> V,
VP.cat=vp,
V.type=finite,
intransitive_verb(V),                                         (2)
VP.voice=active,
verbal_head_features_sharing(VP,V),
VP.number=V.number;

VP -> V NP,
VP.cat=vp,
NP.cat=np,                                                    (3)
V.cat=v,
V.type=finite,
transitive_verb(V),
VP.voice=active,
VP.obj=NP,
VP.number=V.number,
```

---

sont *partagées* entre les fonctions logiques et les fonctions syntaxiques.

```
verbal_head_features_sharing(VP,V),
VP.sem_restr.semconf_of_arg2=NP.semconf;

VP -> AUX V,
VP.cat=vp,
AUX.cat=aux,
be(AUX),
V.cat=v,                                                    (4)
V.type=past_participle,
transitive_verb(V),
VP.voice=passive,
VP.number=AUX.number,
verbal_head_features_sharing(VP,V);

AUX -> is,
be(AUX),                                                    (5)
finite_singular(AUX);

AUX -> are,
be(AUX),                                                    (6)
finite_plural(AUX);

NP -> Det Noun,
NP.cat=np,
Det.cat=det,
Noun.cat=noun,
NP.det=Det.det,
number_agreement(NP,Det),                                   (7)
NP.number=Noun.number,
NP.noun=Noun.noun,
NP.semconf=Noun.semconf;

Noun -> dog,
Noun.noun=dog,
sinnoun(Noun),                                              (8)
high_status_animal(Noun);

Noun -> rats,
Noun.noun=rat,                                              (9)
plunoun(Noun),
high_status_animal(Noun);

Noun -> steaks,
Noun.noun=steak,
plunoun(Noun),                                              (10)
standard_eatable(Noun);

V -> sleep,
intransitive_verb(V),
V.verb=sleep,                                               (11)
animate_arg1(V),
finite_plural(V);

V -> dreams,
intransitive_verb(V),
V.verb=dream,                                               (12)
animate_arg1(V),
finite_singular(V);
```

```
V -> eats,
transitive_verb(V),
animate_arg1(V),
eatable_arg2(V),                                        (13)
V.verb=eat,
finite_singular(V);

V -> eaten,
transitive_verb(V),
animate_arg1(V),
eatable_arg2(V),                                        (14)
V.verb=eat,
past_participle(V);

Det -> a,
Det.cat=det,                                            (15)
Det.det=a,
Det.number=sin;

Det -> the,
Det.cat=det,
Det.det=the,                                            (16)
Det.number=DetNbr,
freeze(DetNbr,sin_or_plu(Det));

Det -> several,
Det.cat=det,                                            (17)
Det.det=several,
Det.number=plu;
```

B Définitions de prédicats sur des DAGS.

```
sinnoun(Noun):
Noun.cat=noun,                                          (1)
Noun.number=sin;

plunoun(Noun):
Noun.cat=noun,                                          (2)
Noun.number=plu;

sin_or_plu(X):
X.number=sin;                                           (3)

sin_or_plu(X):
X.number=plu;

number_agreement(X,Y):
X.number=Y.number;                                      (4)

finite_plural(Verb):
Verb.type=finite,                                       (5)
Verb.number=plu;

finite_singular(Verb):
Verb.type=finite,                                       (6)
Verb.number=sin;
```

```
past_participle(V):
V.type=past_participle;                                              (7)

be(AUX):
AUX.cat=aux,                                                         (8)
AUX.verb=be;

transitive_verb(V):
V.cat=v,                                                             (9)
V.transit=trans;

intransitive_verb(V):
V.cat=v,                                                             (10)
V.transit=int;

verbal_head_features_sharing(X,Y):
X.sem_restr=Y.sem_restr,
X.obj=Y.obj,
X.subj=Y.subj,
X.arg1=Y.arg1,                                                       (11)
X.arg2=Y.arg2,
X.verb=Y.verb,
X.voice=Y.voice;

voice_treatment(S):
S.voice=active,
S.arg1=S.subj,
S.arg2=S.obj,
S.sem_restr.semconf_of_arg1=S.arg1.semconf;                          (12)

voice_treatment(S):
S.voice=passive,
S.arg2=S.subj,
S.sem_restr.semconf_of_arg2=S.arg2.semconf;

animate(X):
X.semconf.animate=yes;                                               (13)

nonanimate(X):
X.semconf.animate=no;                                                (14)

eatable(X):
X.semconf.eatable=yes;                                               (15)

noneatable(X):
X.semconf.eatable=no;                                                (16)

high_status_animal(X):
animate(X),                                                          (17)
noneatable(X);

standard_eatable(X):
eatable(X),                                                          (18)
nonanimate(X);

animate_arg1(V):
V.sem_restr.semconf_of_arg1.animate=yes;                             (19)
```

```
eatable_arg2(V):
V.sem_restr.semconf_of_arg2.eatable=yes;                    (20)
```

2.2.2 Explications concernant la grammaire.

La grammaire de la section précédente est très élémentaire, mais elle illustre les aspects les plus importants du formalisme RATP.

Nous avons subdivisé la grammaire en 2 parties:

- la partie A consiste en un ensemble de "règles hors-contexte augmentées"

- la partie B consiste en un ensemble de "définitions de prédicats sur des DAGs".

Cette subdivision a pour seul but ici de faciliter la lecture. Dans une grammaire RATP, les 2 types de constructions peuvent être arbitrairement mélangées.

2.2.2.1 Les règles de grammaire hors-contexte augmentées

Considérons la règle (1)

```
S -> NP VP,                                          a
S.cat=s,                                             b.1
NP.cat=np,                                           b.2
VP.cat=vp,                                           b.3
number_agreement(NP,VP),                             b.4
S.subj=NP,                                           b.5
verbal_head_features_sharing(S,VP),                  b.6
voice_treatment(S);                                  b.7
```

Dans cette règle, la ligne (a) et les lignes (b.1)...(b.7) jouent des rôles différents:

- la ligne (a) - le "squelette" de la règle - décrit une règle hors- contexte. Ici, S, NP et VP sont des non-terminaux au sens classique.

- les lignes (b.1)...(b.7) - le "corps" de la règle - décrivent des relations (ou contraintes) qui doivent être vérifiées entre des sous-structures (c'est à dire entre des DAGs). Ici, S,N,VP sont des variables qui représentent des DAGs, à savoir les DAGs associés aux non-terminaux S, NP et VP du squelette.

Il faut donc faire attention au fait que les symboles S, NP, VP jouent un double rôle dans une règle de grammaire: dans le squelette, le rôle de non-terminaux, et dans le corps, le rôle de variables désignant des DAGs qui représentent les structures obtenues pour chacun des non-terminaux. Remarquons en passant que dans ce rôle de variables désignant des DAGs (identificateur commençant par une majuscule et apparaissant dans le corps d'une règle), un identificateur peut apparaitre de façon "libre", c'est à dire sans être associé à un non-terminal (par exemble DetNbr dans la règle (16); il joue alors un rôle d'auxiliaire dans la vérification des contraintes.

Considérons maintenant une à une la signification des lignes (b.1) à (b.7) de la règle(1).

(b.1) énonce une contrainte d'égalité. Elle peut se lire de la façon suivante : le sous-DAG (la sous-structure) accédé à partir du DAG associé à S par le chemin (constitué d'une seule étiquette) "S.cat" est égal[5] au DAG (dégénéré c'est à dire réduit à un symbole) "s" (une constante).

Autrement dit, la catégorie du DAG associé à S est "s".

(b.2) et (b.3) se comprennent de façon analogue.

(b.4) exprime une contrainte "prédicative" à 2 arguments, à savoir que la relation "number-agreement" doit être vérifiée entre les 2 DAGS associés à NP et VP. Pour connaitre la spécification de cette

---

[5] ou plutôt "unifiable" (voir Section III).

contrainte, il faut se reporter à la définition (6) du prédicat "number-agreement " dans la partie B de la grammaire. Ce prédicat se lit : number-agreement (X,Y) est vrai lorsque X et Y sont deux DAGs tels que le sous-DAG de X accédé par le chemin "X.number" est égal au sous-DAG de Y accédé par le chemin "Y.number". Autrement dit, le "number" de X est égal au "number" de Y.

(b.5) exprime une contrainte d'égalité : le sous-DAG de S accédé par le chemin S.subj est égal au DAG associé à NP. Autrement dit, le "subj" (le sujet) de S est le NP tout entier (i.e. toute la structure associée à NP).

Si on compare (b.1) à (b.5), on voit que, du point de vue du formalisme, un trait grammatical simple comme "s", ou bien une structure complexe comme la structure associée à un groupe nominal, sont traités de la même façon. Ce sont tous les deux des sous-DAGs de DAGs, l'un dégénéré, l'autre complexe, accédés respectivement par les chemins "S.cat" et "S.subj".

(b.6) exprime une contrainte prédicative (à 2 arguments), à savoir qu'il y a "partage" des "verbal_head_features" (subj, obj, arg1, arg2, verb, voice, sem_restr) entre S et VP.

Le contenu de cette contrainte est explicité dans la définition (définition II dans B) du prédicat "verbal_head_features_sharing".

(b.7) exprime une contrainte prédicative (à 1 argument) "voice_treatment" qui doit être vérifiée par le Dag S. En nous référant à la définition du prédicat "voice-treatment" (B, définition (12)), nous remarquons un aspect nouveau du formalisme : il y a 2 "énoncés" qui correspondent au prédicat "voice-treatment". Ces 2 énoncés peuvent être lus comme une disjonction de conditions: pour que le Dag S vérifie la contrainte voice-treatment, il suffit qu'il vérifie soit le 1er énoncé (cas de la voix active) soit le 2ème énoncé (cas de la voix passive). Dans le 1er cas (voix active) la contrainte impose (entre autres) que

le sous-Dag de S accédé par le chemin S.arg2 soit égal au sous-Dag de S accédé par le chemin S.obj, alors que dans le cas de la voix passive la contrainte impose (entre autre) que le sous-Dag de S accédé par le chemin S.arg2 soit égal au sous-Dag de S accédé par le chemin S.subj. Nous avons montré en section 2.1 des exemples correspondant à ces 2 cas de figure.

2.2.2.2 Les définitions de prédicats.

Les commentaires de la section précédente nous ont déjà fait comprendre le rôle des définitions de prédicat; ces définitions permettent de décrire des contraintes entre DAGs indépendamment d'une règle de grammaire, et aussi:

1) de pouvoir réutiliser la même relation dans plusieurs règles différentes (exemple la définition "transitive-verb" est utilisée dans plusieurs règles de grammaire différentes)

2) d'augmenter considérablement la puissance descriptive du langage (voir section 2.3).

Notons la similarité formelle des définitions de prédicats en RATP avec les définitions de prédicats en PROLOG pur:

- Une définition de prédicat comporte un certain nombre d'énoncés (2 énoncés dans le cas de "voice_treatment")

- Dans un énoncé on distingue la "tête" (exemple: "voice_treatment") du "corps" - ou "queue" - ("S.voice=active, S.arg1=S.subj, S.arg2=S.obj, S.sem_restr.semconf_of_arg1=S.arg1.semconf)

- Les identifications de variables commencent par une majuscule (exemple: "S").

- Les identifications de constantes commencent par une minuscule (exemple: "active")[6]. Cette convention est analogue celle qui est utilisée dans certains dialectes de PROLOG (syntaxe d'Edimbourg).

Remarquons aussi certaines différences:

- Les variables désignent des DAGs, et non des termes.

- Une sous-structure est accédée par un chemin (ex. "S.sem_restr.semconf_of_arg2"), alors qu'en Prolog il n'y a pas de notion explicite d'accès à un sous-terme, mais seulement un accès implicite déterminé par la position d'un sous-terme sous un terme.

2.3 RATP comme formalisme d'écriture de Grammaires et comme langage de programmation.

2.3.1 Ratp comme formalisme d'ecriture de grammaires.

En tant que formalisme d'écriture de grammaires, RATP est un langage qui tient l'exact milieu entre d'une part PATR (représentant de la classe des grammaires d'unification) et d'autre part les DCG (représentant de la classe des grammaires logiques).

Ainsi considérons les trois règles suivantes, extraites de la grammaire de la section 2.2.1:

```
NP -> Det Noun,
NP.cat=np,
Det.cat=det,
```

---

[6] Attention: dans les énoncés de la définition (12), "voice", "subj", "arg1", ..., ne dénotent pas des Dags constants, mais sont des étiquettes. Il ne faut pas confondre ces 2 types de symboles.

```
Noun.cat=noun,
NP.det=Det.det,
number_agreement(NP,Det),
NP.number=Noun.number,
NP.noun=Noun.noun,
NP.semconf=Noun.semconf;

Det -> a,
Det.cat=det,
Det.det=a,
Det.number=sin;

Noun -> dog,
Noun.noun=dog,
sinnoun(Noun),
high_status_animal(Noun);
```

En DCG, un équivalent approximatif pourrait être:

```
np(N,Nbr,X,Det,Y,Z,Semconf) -->
      det(Det,Nbr1), noun(N,Nbr2,Semconf),
      {nbr_agr(np(N,Nbr,X,Det,Y,Z,Semconf),det(Det,Nbr1))} .

det(a,sin) --> [a].

noun(dog,Semconf) -->  [dog],
                  {sinnoun(noun(dog,Semconf))},
                  {high_status_animal(noun(dog,Semconf))}.
```

Ici, comme en RATP, on peut appeler des prédicats qui contraignent les structures construites. En DCG, ces structures sont des termes à "arité" fixe, alors qu'en RATP ce sont des Dags pouvant référer à un nombre indefini a priori de sous-structures.

En PATR, un équivalent approximatif de ces 3 règles serait:

```
NP -> Det Noun,
NP.cat=np,
Det.cat=det,
Noun.cat=noun,
```

```
NP.det=Det.det,
NP.number=Det.number,          % number_agreement(NP,Det)
NP.number=Noun.number,
NP.noun=Noun.noun,
NP.semconf=Noun.semconf;

Det -> a,
Det.cat=det,
Det.det=a,
Det.number=sin;

Noun -> dog,
Noun.noun=dog,
Noun.cat=noun,                          % sinnoun(Noun)
      Noun.number=sin,
                                        % high_status_animal(Noun)
X.semconf.animate=yes,                   % animate(X)
X.semconf.eatable=no;                    % noneatable(X)
```

La différence essentielle avec RATP tient à ce qu'il est impossible de définir des classes de structures à l'aide de prédicats;

De plus, dans le cas où l'on ne restreint pas en RATP l'utilisation d'appels récursifs de prédicats entre eux (cyclicité, voir section suivante), il est généralement impossible de traduire une grammaire RATP dans le formalisme PATR.

Meme dans le cas où la cyclicité est imposée, le mécanisme prédicatif de RATP permet une plus grande modularité dans la définition des structures; il permet aussi de rendre compte de façon naturelle de la disjonction de classes de structures (exemple du prédicat "voice_treatment" dans la grammaire donnée en exemple).

2.3.2 RATP comme langage de programmation.

Nous avons vu que les définitions de prédicats en RATP ont une structure très proche des définitions de prédicats en Prolog, à certaines différences près:

- Dags plutôt que termes
- chemins d' accès

Cela n'est évidemment pas un hasard : la structure de prédicats en RATP a été directement inspirée par la structure de prédicats en Prolog, la différence principale étant due au fait que les arguments d'un prédicat RATP sont des Dags plutôt que des termes: (les conséquences de cette différence sont explorées plus en détail en section 3).

En Prolog, un énoncé (i.e. une "clause de Horn") de type:

$$p(T1,...,Tm) \text{ <-- } q_1(T11, \ldots, T1m_1), \ldots, q_n(Tn1, \ldots, Tnm_n)$$

peut être lu de la façon suivante: si pour tout i ($1 < i \leq n$) le m_i-uplet ($T11,...Tim_i$) appartient à la relation q_i alors le m-uplet ($T1,...,Tm$) appartient à la relation p.

Autrement dit, un programme Prolog peut être vu comme spécifiant, de proche en proche, des classes de n-uplets de termes, à savoir les classes associées à chacun des prédicats du programme: on commence par "remplir" de *ground-terms* les classes correspondant à des prédicats apparaissant dans des énoncés "sans queue" (i.e. vérifiés inconditionnellement), et dès qu'un nouveau terme apparait dans une classe q_i, il a pour conséquence l'apparition de mouveaux termes dans une classe p. Ce processus produit pour chaque prédicat du programme, à la limite, un ensemble de n-uplets de termes qui le satisfont[7].

Ainsi, un programme Prolog peut être vu comme un moyen effectif de décrire et de calculer des classes de structures, ces structures étant ici des termes.

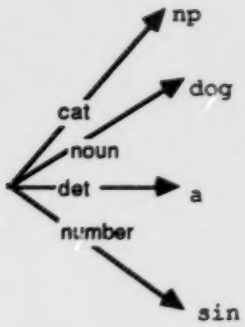
En RATP, le même mécanisme est utilisé : les définitions de prédicats RATP permettent de définir récursivement des classes de DAGs. Ainsi, le prédicat "sin_or_plu(X)" (définition B.3 p.13) permet de définir la

---

[7] le concept formel correspondant à cette description informelle s'appelle le *plus petit point fixe* du programme <LLoyd 1984>.

classe des DAGs dont le "number" a la valeur "sin" ou la valeur "plu"; le prédicat "transitive_verb" (B.9) permet de définir la classe de tous les DAGs qui sont des verbes (i.e. qui ont la catégorie "v") transitifs (qui ont "trans" pour valeur de transit (transitivity).

Ainsi par exemple le Dag A ci-dessous appartient à la classe associée au prédicat sin_or_plu[8]:

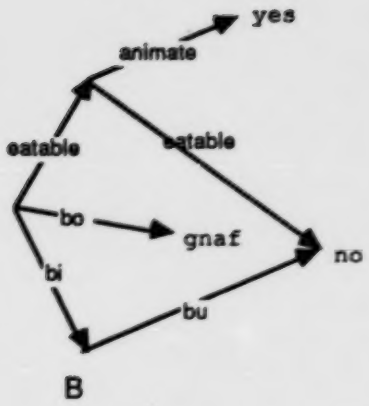


A

Les prédicats peuvent "s'appeler" les uns les autres, ainsi, par exemple, le prédicat high_status_animal (B.17) est défini à l'aide des prédicats "animate" et "noneatable".
Le Dag B ci-dessous est donc dans la classe "high_status_animal":

---

[8] Evidemment, dans le cas de prédicats à plusieurs arguments, on considère des *n-uplets* de Dags.



Le mécanisme de définition "de proche en proche" donne une grande modularité et une grande puissance expressive au langage : dès qu'on a défini une classe de structures, on peut l'utiliser à son tour pour définir de nouvelles classes de structures.

On peut distinguer 2 cas de figure, suivant l'utilisation qui est faite de cette possibilité.

1er cas de figure: ou limite l'expressivité du langage en n'autorisant pas de "cycles" dans les définitions de prédicats (par cycle on entend le fait qu'un prédicat p1 appele un prédicat p2 qui lui-même appelle un prédicat... qui appelle le prédicat p1 dont on est parti).

Si cette limitation est respectée, alors on montre facilement que l'on peut décrire par prédicats exactement les classes de structures que l'on peut décrire sans utiliser de prédicats, uniquement par des équations de chemins.

Autrement dit, si cette limitation est respectée, on ne gagne rien - du point de vue théorique - à autoriser l'utilisation de prédicats dans la grammaire : on pourrait se contenter de règles de grammaire augmentées

par des équations[9]; mais on gagne cependant beaucoup au niveau pratique du fait de la modularité et de la réutilisabilité apportées par l'utilisation de prédicats.

2ème cas de figure: on admet des cycles dans les définitions de prédicats.

Dans ce cas, un prédicat peut s'appeler lui-même, soit directement, soit par l'intermédiaire d'autres prédicats. C'est la situation du langage Prolog, et c'est cette possibilité qui fait de Prolog un langage de programmation universel, c'est à dire que n'importe quelle classe de termes "récursivement énumérable" peut être décrite par un prédicat de ce type.

De la même façon, si on admet des cycles dans les définitions de prédicats en RATP, on augmente considérablement la puissance descriptive du langage, puisqu'on peut alors décrire par un prédicat RATP n'importe quelle classe de Dags récursivement énumérable[10].

Ainsi, à titre d'exemple, on peut facilement "programmer" des algorithmes d'addition et de multiplication en RATP. Un tel programme est présenté en Annexe 2; ce programme définit un prédicat mult(D) à un seul argument qui est vérifié par une certaine classe de Dags, à savoir les Dags qui représentent, dans une certaine convention de codage des nombres, des "multiplications".

---

[9] C'est le cas de PATR II, au niveau de la grammaire, mais PATR II introduit certains mécanismes qui étendent ses possibilités, au niveau lexical, par exemple.

[10] En toute rigueur, ceci est inexact: en réalité, on peut ainsi décrire n'importe quelle classe récursivement énumérable de Dags qui a la propriété d'être *libre en dehors d'un ensemble fini d'étiquettes*, c'est à dire telle que les étiquettes du complément d'un certain ensemble fini d'étiquettes puissent prendre n'importe quelle valeur dans la classe. Mais nous n'entrerons pas dans ces détails ici.

Par exemple, le Dag A suivant représente une multiplication (celle de 3 par 2) et vérifie le prédicat mult(A).

```
arg1:
    car:
        one
    cdr:
        car:
            one
        cdr:
            car:
                one
            cdr:
                []
arg2:
    car:
        one
    cdr:
        car:
            one
        cdr:
            []
result:
    car:
        one
    cdr:
        car:
            one
        cdr:
            car:
                one
            cdr:
                car:
                    one
                cdr:
                    car:
                        one
                    cdr:
                        car:
                            one
                        cdr:
                            car:
                                one
                            cdr:
                                []
```

De la même façon qu'en Prolog, l'évaluation d'un prédicat en RATP a pour conséquence soit la vérification, soit l'énumération de tous les Dags[11] qui vérifient le prédicat, dépendant du degré d'instanciation du Dag qui est passé initialement en argument au prédicat. Cette évaluation est généralement faite lors de l'appel du prédicat par une règle de la grammaire, mais peut aussi être effectuée indépendamment de la grammaire.

3 DAGS ET TERMES. QUESTIONS D'IMPLANTATION.

3.1 Dags et Termes.

Comme nous l'avons vu dans la section précédente, RATP est un formalisme intermédiaire entre les Grammaires d'Unification, (plus spécifiquement PATR II), d'une part, et d'autre part les Grammaires Logiques (plus spécifiquement les DCG).

De plus, nous avons vu que l'aspect "langage de programmation" de RATP est fortement inspiré de la base théorique de Prolog, la différence essentielle étant due au fait que les structures manipulées sont des Dags plutôt que des termes.

Le choix de Prolog comme langage d'implantation de RATP s'impose donc à l'esprit tout naturellement.

Un des problèmes qui doivent être resolus dans une telle implantation est le problème de la représentation des Dags dans le langage Prolog.

Plusieurs choix sont possibles à priori, - par exemple on pourrait représenter un Dag comme un ensemble de relations à 3 arguments

---

[11] c'est à dire, bien entendu, pas l'énumération de tous les *ground* Dags, mais l'énumération d'un ensemble de dags variables qui couvrent collectivement l'ensemble de toutes les solutions.

"arc(Noeud1,Noeud2,ArcLabel)" et réaliser l'unification de deux Dags en manipulant ces représentations explicites de graphes -, mais il est beaucoup plus souhaitable, pour des raisons d'efficacité, d'utiliser - si possible - le mecanisme d'unification de termes "built-in" de Prolog.

Cela suppose, cependant, que l'on puisse encoder les Dags comme des termes de telle facon que l'unification de 2 Dags se ramène a une unification de termes.[12]

3.2 Implantation des Dags comme des Termes.

De ce point de vue, certaines méthodes naives d'encodage de Dags ne fonctionneront pas; Ainsi, par exemple, on pourrait vouloir encoder un Dag comme une liste[13] de couples de la forme (label_i, dag_i), ou chaque dag_i est à son tour une structure du même type; On pourrait alors definir l'unification de 2 dags comme une sorte d'"union" des 2 listes telle que si un label apparait dans une seule des 2 listes, alors le couple associé est "ajouté" à la liste resultat, et si le même label_i1=label_j2 apparait dans les deux listes l1 et l2, alors le dag associé à ce label dans la liste résultat est l'unification récursive des (listes représentant les) dags dag_i1 et dag_i2.

Malheureusement, comme on peut facilement s'en convaincre, cette méthode ne respecte pas une condition nécessaire à l'unification de Dags:

> Si à un moment donné t1 le Dag A a été unifié avec le Dag B, et si plus tard (temps t2) A a été unifié avec C, et (temps t3) B avec

---

12 Dans ce rapport, nous ne décrivons pas la notion d'unification de termes, notion pour laquelle nous renvoyons le lecteur par exemple à <Giannesini et al. 86>, pas plus que celle, moins connue, d'unification de Dags, pour laquelle on pourra se référer à <Shieber 86>.

13 c'est-à-dire un terme d'une forme particulière.

D, alorscela doit impliquer que C et D sont unifiables au temps t4.

Deux méthodes différentes peuvent cependant être proposées pour résoudre ce problème.

3.2.1 Encodage des Dags comme des listes incomplètes.

Pour résoudre le problème précédent, Eisele et Doerre <Eisele et al. 86> ont proposé une méthode ingénieuse d'encodage des Dags.

Dans leur approche, un Dag est encodé comme une liste Prolog incomplète L de la forme:

L: C1.C2. ... .Cn.X ,

où chaque Ci est un couple de la forme

c(Label,Value)

où récursivement Value est une liste incomplète (de la même forme que L) représentant un Dag.

Alors, l'unification de deux Dags représentés par L1 et L2 est effectuée à l'aide de l'appel au prédicat Prolog

dag_unify(L1,L2)

qui est défini de la façon suivante:

```
dag_unify(L,L): !.
dag_unify(c(Label,V1).R1,L2):
      delete(c(Label,V2),L2,R2),
      dag_unify(V1,V2),
      dag_unify(R1,R2).

delete(C,C.L,L): ! .
delete(C,CC.L1,CC.L2): delete(C,L1,L2).
```

Le prédicat dag_unify a la propriété remarquable de posséder l'invariant suivant:

après un dag_unify réussi de L1 et L2, L1 et L2 ont la forme:

L1: C11.C21. ... .Cn1.X
L2: C12.C22. ... .Cn2.X

où C11.C21. ... .Cn1 et C12.C22. ... .Cn2 sont une "permutation" l'un de l'autre, où par permutation il faut entendre: -a) que les listes des etiquettes Et11 Et21 ... Etn1 (resp. Et12 Et22 ... Etn2) qui sont les premiers éléments des couples Ci1 (resp. Ci2) sont despermutations l'une de l'autre et - b) que les valeurs Vi1 et Vj2 associées à la meme etiquette dans L1 et L2 ont ete elle meme "dag-unifiees" et donc respectent récursivement le meme invariant.

De cela il découle que L1 et L2 représentent le même dag ET qu'ILS GARDERONT CETTE PROPRIETE DANS TOUTE LEUR HISTOIRE PROLOG SUBSEQUENTE, puisque X est commune aux deux objets. Bien évidemment L1 et L2 ne sont cependant pas unifiables EN TANT QUE termes Prolog.

Cette propriété garantit que l'unification de dags a bien les qualités requises. Elle a cependant les inconvénients:

- d'être coûteuse en temps de calcul (parcours de listes)
- de creer une multiplication de termes Prolog différents (et non unifiables en tant que termes Prolog), qui représentent tousle même Dag.

C'est pourquoi nous lui avons préféré la méthode suivante, qui a l'avantage de ramener l'unification des Dags à une simple unification Prolog des termes qui les représentent.

3.2.2 Encodage des Dags comme des arbres ternaires incomplets.

Afin de pouvoir traiter l'unification de dags comme une unification directe de termes Prolog, nous avons associé à chaque label utilisé dans la grammaire une codification sous forme de liste binaire (c'est-a-dire une liste de 0 et de 1) identifiant de facon unique ce label[14].

Supposons maintenant que nous voulions par exemple encoder le Dag

suj: Value1
obj: Value2
cat: Value3

(où Value1, Value2 et Value3 sont eux-mêmes récursivement des encodages de Dags que nous ne donnons pas ici explicitement).

Supposons de plus que les étiquettes suj, obj et cat soient respectivement encodées comme les listes binaires:

suj 0.1.1.nil
obj 1.nil
cat nil

---

[14] L'association de listes binaires aux labels est faite actuellement "at compile time", mais rien n'empêche de la faire "at run time" et d'ajouter dynamiquement de nouvelles étiquettes.

alors le Dag ci-dessus sera encodé comme le terme:

```
b(Value3,
  b(X1,
    X2,
    b(X3,
      X4,
      b(Value1,
        X5,
        X6)))
  b(Value2,X7,X8))
```

où X1...X8 sont des variables non instanciées.

Cette structure se comprend facilement si l'on prend en compte la description informelle suivante:

```
b(Value,          on est "arrivé", i.e. la
                  liste binaire est reduite à nil.

  Gauche,         le premier élément de la liste binaire est 0,
                  on "tourne" à gauche.

  Droite,         le premier élément de la liste binaire est 0,
                  on "tourne" à droite.
 )
```

Cette codification garantit que l'unification de Dags se ramène à l'unification directe (au sens Prolog) des termes qui les représentent.

ANNEXE 1 : Une grammaire RATP plus complexe.

(traduction en RATP de la grammaire 3 de <Shieber 86>)

ANNEXE 1.1 : la grammaire

```
S -> NP VP,
cat_s(S),
cat_np(NP),
cat_vp(VP),
S.head=VP.head,
S.head.form=finite,
VP.syncat.first=NP,
VP.syncat.rest=end;

VP -> V,
cat_vp(VP),
V.cat=v,
VP.head=V.head,
VP.syncat=V.syncat;

VP -> VP1 X,
cat_vp(VP),
cat_vp(VP1),
VP.head=VP1.head,
VP1.syncat.first=X,
VP1.syncat.rest=VP.syncat;

VP1 -> VP,
VP1=VP;

X -> VP,
X=VP;

X -> NP,
X=NP;

NP -> uther,
cat_np(NP),
ms(NP),
third(NP),
NP.head.trans=uther;

NP -> cornwall,
cat_np(NP),
ms(NP),
third(NP),
NP.head.trans=cornwall;

NP -> knights,
cat_np(NP),
mp(D),
third(NP),
NP.head.trans=knights;

V -> sleeps,
finite_3s(V),
sc_np(V),
V.head.trans.pred=sleep;

V -> sleep,
finite_sauf_3s(V),
sc_np(V),
V.head.trans.pred=sleep;
```

```
V -> sleep,
cat_v(V),
V.head.form=nonfinite,
sc_np(V),
V.head.trans.pred=sleep;

V -> storms,
finite_3s(V),
sc_np_np(V),
V.head.trans.pred=storm;

V -> stormed,
pastparticiple(V),
sc_np_np(V),
V.head.trans.pred=storm;

V -> storm,
nonfinite(V),
sc_np_np(V),
V.head.trans.pred=storm;

V -> has,
finite_3s(V),
have(V);

V -> have,
finite_sauf_3s(V),
have(V);

V -> persuades,
V.cat=v,
V.head.form=finite,
allow_type_control(V),
V.head.trans.pred=persuade;

V -> persuaded,
V.cat=v,
V.head.form=pastparticiple,
allow_type_control(V),
V.head.trans.pred=persuade;

V -> promises,
V.cat=v,
V.head.form=finite,
promise_type_control(V),
V.head.trans.pred=promise;

V -> promised,
V.cat=v,
V.head.form=pastparticiple,
promise_type_control(V),
V.head.trans.pred=promise;

V -> to,
V.cat=v,
V.head.form=infinitival,
V.syncat.first.cat=vp,
```

```
V.syncat.first.head.form=nonfinite,
V.syncat.first.syncat.rest=end,
V.syncat.first.syncat.first=V.syncat.rest.first,
V.syncat.rest.first.cat=np,
V.syncat.rest.rest=end,
V.head.trans.pred=V.syncat.first.head.trans;

cat_s(X):
X.cat=s;

cat_vp(X):
X.cat=vp;

cat_np(X):
X.cat=np;

cat_v(X):
X.cat=v;

ms(NP):
NP.head.agreement.gender=masculine,
NP.head.agreement.number=singular;

mp(NP):
NP.head.agreement.gender=masculine,
NP.head.agreement.number=plural;

third(NP):
NP.head.agreement.person=third;

nonfinite(V):
V.cat=v,
V.head.form=nonfinite;

finite_3s(V):
cat_v(V),
V.head.form=finite,
suj_of_v(V,SUJ),
SUJ.head.agreement.person=third,
SUJ.head.agreement.number=singular;

finite_sauf_3s(V):
cat_v(V),
V.head.form=finite,
suj_of_v(V,SUJ),
not_3s(SUJ);

not_3s(SUJ):
SUJ.head.agreement.person#third,
SUJ.head.agreement.number=singular;

not_3s(SUJ):
SUJ.head.agreement.person=third,
SUJ.head.agreement.number#singular;

suj_of_v(V,SUJ):
V.syncat=L,
V.syncat.rest=X,
```

```
freeze(X,suj_of_v_aux(V,SUJ,L));

suj_of_v_aux(V,SUJ,L):
L.rest=end,
L.first=SUJ;

suj_of_v_aux(V,SUJ,L):
L.rest=end,
L.rest=LL,
suj_of_v_aux(V,SUJ,LL);

pastparticiple(V):
V.cat=v,
V.head.form=pastparticiple;

pastpart(V):
V.head.form=pastparticiple;

have(V):
V.syncat.first.cat=vp,
V.syncat.first.head.form=pastparticiple,
V.syncat.first.syncat.rest=end,
V.syncat.first.syncat.first=V.syncat.rest.first,
V.syncat.rest.first.cat=np,
V.syncat.rest.first.head.agreement.person=third,
V.syncat.rest.first.head.agreement.number=singular,
V.syncat.rest.rest=end,
V.head.trans.pred=perfective,
V.head.trans.arg1=V.syncat.first.head.trans;

allow_type_control(V):
cat_v(V),
sc_np_np_tovp(V),
V.syncat.rest.first.syncat.first=V.syncat.first;

promise_type_control(V):
cat_v(V),
sc_np_np_tovp(V),
V.syncat.rest.first.syncat.first=V.syncat.rest.rest.first;


sc_np(V):
V.syncat.first.cat=np,
V.syncat.rest=end,
V.head.trans.arg1=V.syncat.first.head.trans;

sc_np_np(V):
V.syncat.first.cat=np,
V.syncat.rest.first.cat=np,
V.syncat.rest.rest=end,
V.head.trans.arg1=V.syncat.rest.first.head.trans,
V.head.trans.arg2=V.syncat.first.head.trans;

sc_np_np_tovp(V):
V.syncat.first.cat=np,
V.syncat.rest.first.cat=vp,
V.syncat.rest.first.head.form=infinitival,
```

```
V.syncat.rest.first.syncat.rest=end,
V.syncat.rest.rest.rest=end,
V.head.trans.arg1=V.syncat.rest.rest.first.head.trans,
V.head.trans.arg2=V.syncat.first.head.trans,
V.head.trans.arg3=V.syncat.rest.first.head.trans;
```

ANNEXE 1.2 : quelques phrases analysées par cette grammaire

```
52 /ul/dta/dymetman> ratpi

                        MPROLOG (2.1.0) LOGIC - LAB
                           (c) 1985 Logicware Inc.
                    PDSS Program Development Support System


                              RATP INTERPRETER

Which grammar file ? :
*s8

s8.ratp_output LOADED .

;;;;;;;;;;;;;; quelques analyses de la "classe" S

: ?parse(s,uther.storms.cornwall.nil).

[uther,storms,cornwall]

 cat:
   * s
 head:
     form:
         finite
     trans:
         arg1:
             uther
         arg2:
             cornwall
         pred:
             storm

Yes

: ?parse(s,uther.has.promised.knights.to.storm.cornwall.nil).

[uther,has,promised,knights,to,storm,cornwall]

 cat:
     s
 head:
     form:
         finite
     trans:
         arg1:
             arg1:
                 uther
             arg2:
                 knights
             arg3:
                 pred:
                     arg1:
                         uther
                     arg2:
                         cornwall
```

```
                    pred:
                        storm
                pred:
                    promise
            pred:
                perfective


Yes

;;;; on remarquera la différence entre promise et persuade: c'est l'arg1
;;;; de promise qui fait l'action, alors que c'est l'arg2 de persuade
;;;; qui fait l'action.

: ?parse(s,uther.has.persuaded.knights.to.storm.cornwall.[]).

[uther,has,persuaded,knights,to,storm,cornwall]

 cat:
     s
 head:
     form:
         finite
     trans:
         arg1:
             arg1:
                 uther
             arg2:
                 knights
             arg3:
                 pred:
                     arg1:
                         knights
                     arg2:
                         cornwall
                     pred:
                         storm
             pred:
                 persuade
         pred:
             perfective


Yes



;;;;;;;;;;;;;; quelques "analyses" de la "classe" NP
;;;;;;;;;;;;;; L est une variable ici et donc c'est reellement la
synthese de
;;;;;;;;;;;;;; tous les np qui est effectuee.

: ?parse(np,L).

[uther]

 cat:
     np
```

```
 head:
      agreement:
          gender:
              masculine
          number:
              singular
          person:
              third
      trans:
          uther


      L = [uther]
Continue (y/n) ?

[cornwall]

 cat:
      np
 head:
      agreement:
          gender:
              masculine
          number:
              singular
          person:
              third
      trans:
          cornwall

      L = [cornwall]
Continue (y/n) ?

[knights]

 cat:
      np
 head:
      agreement:
          person:
              third
      trans:
          knights


      L = [knights]
Continue (y/n) ?
NO

;;;;;;;;;;;;;; quelques "analyses" de la "classe" VP
;;;;;;;;;;;;;; L est une variable ici et donc c'est reellement la
synthese
;;;;;;;;;;;;;; des vp qui est effectuee.

: ?parse(vp,L).

[sleeps]
```

```
 cat:
     vp
 head:
     form:
         finite
     trans:
         pred:
             sleep
 syncat:
     first:
         cat:
             np
         head:
             agreement:
                 number:
                     singular
                 person:
                     third
     rest:
         end


     L = [sleeps]
Continue (y/n) ?

[sleep]

 cat:
     vp
 head:
     form:
         finite
     trans:
         pred:
             sleep
 syncat:
     first:
         cat:
             np
         head:
             agreement:
                 number:
                     singular
     rest:
         end


     L = [sleep]
Continue (y/n) ?

[sleep]

 cat:
     vp
 head:
     form:
         finite
```

```
    trans:
        pred:
            sleep
syncat:
    first:
        cat:
            np
        head:
            agreement:
                person:
                    third
    rest:
        end

    L = [sleep]
Continue (y/n) ?

[sleep]

cat:
    vp
head:
    form:
        nonfinite
    trans:
        pred:
            sleep
syncat:
    first:
        cat:
            np
        head:
    rest:
        end


    L = [sleep]
Continue (y/n) ?

[storms]

cat:
    vp
head:
    form:
        finite
    trans:
        pred:
            storm
syncat:
    first:
        cat:
            np
        head:
    rest:
        first:
            cat:
```

```
            np
        head:
            agreement:
                number:
                    singular
                person:
                    third
    rest:
        end

L = [storms]
Continue (y/n) ?  n
OK
;;;;;;;;;;;;;; une analyse de la "classe" VP

: ?parse(vp,has.persuaded.knights.to.storm.cornwall.nil).

[has,persuaded,knights,to,storm,cornwall]

cat:
    vp
head:
    form:
        finite
    trans:
        arg1:
            arg2:
                knights
            arg3:
                pred:
                    arg1:
                        knights
                    arg2:
                        cornwall
                    pred:
                        storm
            pred:
                persuade
        pred:
            perfective
syncat:
    first:
        cat:
            np
        head:
            agreement:
                number:
                    singular
                person:
                    third
    rest:
        end


Yes
: bye
```

```
*** The following module(s) have not been saved: ***
        unnamed_module
Do you want to exit (y/n) ?  y
Normal exit from MPROLOG PDSS
52 /ul/dta/dymetman> ^D
script done on Fri Mar 27 19:26:35 1987
```

ANNEXE 2 : Un programme de multiplication en RATP

```
add(D):
D.arg1=nil,
D.arg2=D.result;

add(D):
D.arg1.car=one,
D.arg1.cdr=D1.arg1,
D.arg2=D1.arg2,
D.result.car=one,
D.result.cdr=D1.result,
add(D1);


mult(D):
D.arg1=nil;
D.result=nil;

mult(D):
D.arg1.car=one,
D.arg1.cdr=X,
D.arg2=Y,
D.result=Z,
D1.arg1=X,
D1.arg2=Y,
D1.result=Z1,
mult(D1),
D2.arg1=Y,
D2.arg2=Z1,
D2.result=Z,
add(D2);


mm(D):
D.arg2.car=one,
D.arg2.car=one,
D.arg2.cdr.car=one,
D.arg2.cdr.cdr=nil,
D.arg1.car=one,
D.arg1.cdr.car=one,
D.arg1.cdr.cdr.car=one,
D.arg1.cdr.cdr.cdr=nil,
mult(D);
```

BIBLIOGRAPHIE

Bresnan,J and Kaplan, R, 1983: "Lexical-Functional Grammar: a Formal System for Grammatical Representation", in J. Bresnan (ed.), *The mental representation of grammatical relations,* MIT Press.

Colmerauer, A, 1978: "Metamorphosis Grammars", in L.Bolc (ed) Lecture Notes in Computer Science, Springer Verlag, vol 63.

Dahl, V, 1985: "Logic Based Metagrammars for Natural Language Analysis", Technical Report, Computing Science, Simon Fraser University, Vancouver.

Eisele, A, and Dörre, J, 1986: "A Lexical-Functional System in Prolog", Proceedings of Coling 86.

Giannesini, F, Kanoui,H, Pasero, R, Van Caneghem, M, 1985, "PROLOG", Interéditions, Paris.

Kay, M, 1983, "Unification Grammar", Xerox Palo Alto Research Unit, Palo Alto.

Lloyd, J, 1984: "Foundations of Logic Programming", Springer Verlag.

Pereira, F, 1981: "Extraposition Grammars", American Journal of Computational Linguistics, vol. 9:3.

Pereira, F, and Warren, 1980: "Definite Clause Grammars for Language Analysis", Artificial Intelligence, vol.13,.

Shieber, S, 1984: "The Design of a Computer Language for Linguistic Information", Proceedings of the ACL 1984.

Shieber,S, 1986: "An Introduction to Unification-Based Grammar Formalisms", CSLI, Stanford.